Asignatura: Fundamentos de algoritmia
Evaluación continua algoritmos iterativos. Cuestionario.

Grupo A.
Profesor: Isabel Pita.

Nombre del alumno:

1. Las condiciones que debe verificar un bucle: `{P} while (B) { A } {Q}` para ser correcto son:

   a) $P \Rightarrow I$ b) $\{I \wedge B\}A\{I\}$ c) $I \wedge \neg B \Rightarrow Q$
   d) $I \wedge B \Rightarrow t > 0$ e) $\{I \wedge B \wedge t = T\}A\{t < T\}$

   donde I es un predicado invariante y t una función cota.

   Explica lo que prueba cada una de estas condiciones.

   Respuesta:

   Las tres primeras condiciones a), b), c) se refieren a la corrección de las instrucciones y expresiones del bucle. Las dos últimas d) y e) se refieren a la terminación del bucle.

   La condición a) comprueba que el predicado invariante se cumpla antes de comenzar el bucle.

   La condición b) comprueba que el invariante se cumpla después de ejecutar cada vuelta del bucle.

   La condición c) comprueba que cuando acaba el bucle se cumpla la postcondición.

   La condición d) exige que la función cota sea positiva cuando se ejecuta el bucle.

   La condición e) comprueba que la función cota disminuya en cada vuelta del bucle.

2. Una función cota t debe cumplir que: d) $I \wedge B \Rightarrow t > 0$ y e) $\{I \wedge B \wedge t = T\}A\{t < T\}$. Dado el bucle

```
std::vector<int> v1, v2, sol; ......
int i = 0; int j = 0;
while (i < v1.size() && j < v2.size()) {
    if (v1[i] < v2[j]) { sol.push_back(v1[i]); ++i; }
    else if (v1[i] > v2[j]) { sol.push_back(v2[j]); ++j; }
    else { sol.push_back(v1[i]); ++i; ++j;  }
}
```

   - Indica el problema que resuelve el programa.
   - Da una función cota que permita probar la terminación del bucle y explica porqué cumple las condiciones anteriores.

   Respuesta:

   Dados dos vectores ordenados el algoritmo hace una mezcla ordenada en el vector solución de los dos vectores hasta que termina uno de ellos. Si los vectores no están ordenados el algoritmo copia valores de uno u otro según como estén distribuidos en v1 y v2.

   Una función cota es $t(i) = v1.size - i + v2.size - j$. Es positiva porque en el bucle la variable `i` es positiva y no alcanza el valor de $v1.size$ y la variable `j` es positiva y no alcanza el valor de $v2.size$. En cada vuelta del bucle el valor de la función cota decrece ya que o se incrementa `i` o se incrementa `j`.

3. Dada la especificación:

   P: $\{v.size() \geq 0\}$
   `resolver(vector<int> v)` dev `int np`
   Q: $\{np == \text{máx } p, q : 0 <= p <= q <= v.size() \wedge positivos(v, p, q) : q - p)\}$

   donde *positivos(v,p,q)* es un predicado que indica que todos los valores del vector entre las posiciones p y q son positivas.

   Y dada la siguiente implementación:

```
int resolver(std::vector<int> const& v) {
    int np = 0; int longAct = 0;
    for (int i = 0; i < v.size(); ++i) {
        if (v[i] > 0) { // El elemento continua la racha
            ++longAct;
            if (np < longAct) np = longAct;
        }
        else longAct = 0; // Se rompe la racha
    }
    return np;
}
```

Indica un invariante del bucle que nos permita probar la corrección del mismo. Recuerda que en el invariante se debe expresar lo que se calcula en todas las variables que intervienen en el bucle, no solo en la variable que se devuelve como resultado de la función.

Respuesta:

I: $\{np == \text{máx } p,q : 0 <= p <= q <= i \wedge positivos(v,p,q) : q-p) \wedge longAct = max\ p : 0 < p < i : \forall k : p <= k < i \wedge positivos(v,p,i) : i - p \wedge 0 <= i <= v.size\}$

La primera parte del invariante expresa que la variable `np` del programa tiene como valor la longitud del segmento máximo de números positivos de la parte ya recorrida por el bucle. La segunda parte indica que la variable `longAct` tiene como valor la longitud del máximo segmento que termina en `i` y es todo el de valores positivos. La tercera parte indica el rango de valores que puede tomar la variable de control del bucle `i` durante la ejecución del bucle.

4. Indica el coste del siguiente algoritmo que también resuelve el problema anterior. Justifica tu respuesta indicando cuantas vueltas dan en total los bucles.

```
int resolver(std::vector<int> const& v) {
    int np = 0;  int i = 0;
   while (i < v.size()) {
        while (i < v.size() && v[i] <= 0) ++i;
        int cont = 0;
        while (i < v.size() && v[i] > 0) { ++cont; ++i; }
        if (cont > np) np = cont;
    }
    return np;
}
```

Respuesta:

El algoritmo tiene coste $\mathcal{O}(\backslash)$ siendo n el número de elementos del vector.

El bucle de fuera, en el peor caso da v.size() vueltas. Si nos fijamos en los dos bucles del cuerpo del bucle while externo, vemos que por cada vuelta que da uno de estos bucles internos se evita una vuelta del bucle externo, ya que la variable de control de estos bucles es la misma que la del bucle externo. Por lo tanto entre todos los bucles se dan n vueltas siendo n el tamaño del vector.

5. Dada la siguiente especificación:

P:$\{true\}$
resolver(std::vector<int> & v) dev `bool b`
Q:$\{b \equiv (\sum w : 0 \leq w < v.size : v[w]) > 0\}$

implementada mediante las dos funciones siguientes:

```
bool resolver(std::vector<int> const& v) {
  int i = 0; int suma = 0;
  while (i < v.size()) {
    suma += v[i];
    ++i;
  }
  return suma > 0;
}
```

```
bool resolver(std::vector<int> const& v) {
  int j = v.size(); int suma = 0;
  while (0 < j) {
    suma += v[j-1];
    --j;
  }
  return suma > 0;
}
```

a) Indica un invariante del bucle izquierdo que nos permita probar la corrección del mismo. Justifica que el invariante propuesto cumple a) $P \Rightarrow I$ y c) $I \wedge \neg B \Rightarrow Q$.

b) Indica un invariante del bucle derecho y marca las diferencias con el invariante del bucle izquierdo.

Respuesta:

a) Invariante I:$\{suma = \sum w : 0 \leq w < i : v[w] \wedge 0 \leq i \leq v.size\}$.

Se cumple $P \Rightarrow I$. La precondición del bucle es: $P : \{i == 0 \wedge suma = 0\}$. Si $i == 0$ el rango del sumatorio es vacío y por lo tanto su valor cero. La variable `suma` toma el valor cero, por lo que se cumple en invariante.

Se cumple $I \wedge \neg B \Rightarrow Q$. En el invariante se indica $i \leq v.size$ y la negación de la condición es $i \geq v.size$. De estas dos condiciones se deriva que $i == v.size$. Imponiendo esta condición en el invariante se obtiene que $suma = \sum w : 0 \leq w < v.size : v[w]$, que es la postcondición del algoritmo.

b) Invariante I:$\{suma = \sum w : j \leq w < v.size : v[w] \wedge 0 \leq j < v.size\}$.

La diferencia entre los dos invariantes está en los límites de las variables ligadas al sumatorio, que indican el rango de valores sumados. En el primer caso se han sumado los valores de la izquierda de la variable de control del bucle y en el segundo caso se han sumado los valores a la derecha de la variable de control del bucle.

6. El algoritmo de partición separa los elementos de un vector en dos o tres partes, dependiendo de la versión que se utilice, dejando en cada una de las partes los elementos que cumplen una determinada propiedad. Por ejemplo, deja los elementos menores que un cierto valor en la parte izquierda del vector, los elementos iguales en la parte central y los elementos mayores en la parte derecha. Escribe en pseudocódigo cómo se implementa este algoritmo. Puedes elegir la versión que divide en dos partes o la versión que divide en tres partes. Indica el invariante y la función cota.

Respuesta: Versión que divide el vector en dos partes: (problema *Viajes a Martes* del cuaderno de problemas de iterativos del campus)

```
int particion (std::vector<int> & v, int pivote) {
    int p = 0, q = (int)v.size()-1;
    while (p <= q) {
        if (v[p] <= pivote)  ++p; // elemento del indice izquierdo correcto
        else if (v[q] > pivote) --q;  // elemento del indice derecho correcto
        else {  // Ambos elementos fuera de sitio
            std::swap(v[p],v[q]);
            ++p; --q;
        }
    }
    return p;
}
```

Invariante: $I \equiv \forall k : 0 \leq k < p : v[k] <= pivote \wedge \forall k : q < k < v.size : v[k] > pivote$

Función cota: t(i,j) = j - i + 1

Versión que divide el vector en tres partes: (problema *Cintas de colores* del cuaderno de problemas de iterativos del campus)

```
void particion (std::vector<int> & v, int &p, int &q, int pivote) {
    p = 0; q = (int)v.size()-1; int k = 0;
    while (k <= q ) {
        if (v[k] == pivote) ++k;
        else if ( v[k] < pivote)  {
            std::swap(v[p],v[k]);
            ++p; ++k;
        }
        else {
            std::swap(v[q],v[k]);
            --q;
        }
    }
}
```

Invariante: $I \equiv \forall x : 0 \leq x < p : v[x] < pivote \wedge \forall x : p \leq x <= q : v[x] == pivote \wedge \forall x : q < x < v.size : v[x] > pivote$

Función cota: t(k,q) = q - k + 1